APPENDIX A

Memory Allocation in C Programs

C supports three kinds of memory allocation through the variables in C programs:

Static allocation When we declare a static or global variable, static allocation is done for the variable. Each static or global variable is allocated a fixed size of memory space. The number of bytes reserved for the variable cannot change during execution of the program.

Automatic allocation When we declare an automatic variable, such as a function argument or a local variable, automatic memory allocation is done. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when it exits from a compound statement.

Dynamic allocation A third important kind of memory allocation is known as *dynamic allocation.* In the following sections we will read about dynamic memory allocation using pointers.

Memory Usage

Before jumping into dynamic memory allocation, let us first understand how memory is used. Conceptually, memory is divided into two parts—program memory and data memory (Fig. A1).

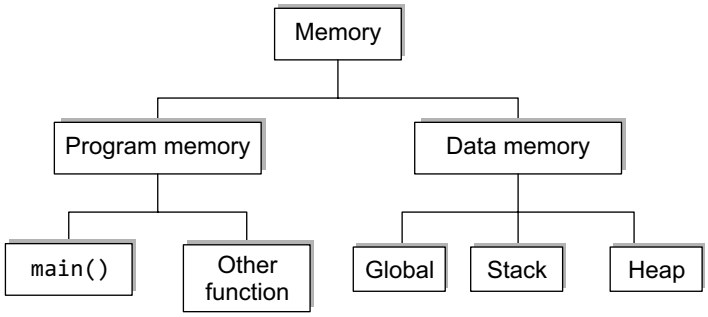

Figure A1 Memory usage

The program memory consists of memory used for the `main()` and other called functions in the program, whereas data memory consists of memory needed for permanent definitions such as global data, local data, constants, and dynamic memory data. The way in which `C` handles the memory requirements is a function of the operating system and the compiler.

When a program is being executed, its `main()` and all other functions are always kept in the memory. However, the local variables of the function are available in the memory only when they are active. When we studied recursive functions, we have seen that the system stack is used to store a single copy of the function and multiple copies of the local variables.

Apart from the stack, we also have a memory pool known as heap. Heap memory is unused memory allocated to the program and available to be assigned during its execution. When we dynamically allocate memory for variables, heap acts as a memory pool from which memory is allocated to those variables.

However, this is just a conceptual view of memory and implementation of the memory is entirely in the hands of system designers.

Dynamic Memory Allocation

The process of allocating memory to the variables during execution of the program or at run time is known as *dynamic memory allocation*. C language has four library routines which allow this function.

Till now whenever we needed an array we had declared a static array of fixed size as

```
int arr[100];
```

When this statement is executed, consecutive space for 100 integers is allocated. It is not uncommon that we may be using only 10% or 20% of the allocated space, thereby wasting rest of the space. To overcome this problem and to utilize the memory efficiently, C language provides a mechanism of dynamically allocating memory so that only the amount of memory that is actually required is reserved. We reserve space only at the run time for the variables that are actually required. Dynamic memory allocation gives best performance in situations in which we do not know memory requirements in advance.

C provides four library routines to automatically allocate memory at the run time. These routines are shown in Table A1.

Table A1 Memory allocation/de-allocation functions

Function	Task
malloc()	Allocates memory and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
free()	Frees previously allocated memory
realloc()	Alters the size of previously allocated memory

When we have to dynamically allocate memory for variables in our programs then pointers are the only way to go. When we use malloc() for dynamic memory allocation, then you need to manage the memory allocated for variables yourself.

Memory Allocations Process

In computer science, the free memory region is called heap. The size of heap is not constant as it keeps changing when the program is executed. In the course of program execution, some new variables are created and some variables cease to exist when the block in which they were declared is exited. For this reason it is not uncommon to encounter memory overflow problems during dynamic allocation process. When an overflow condition occurs, the memory allocation functions mentioned above will return a null pointer.

Allocating a Block of Memory

Let us see how memory is allocated using the malloc()function. malloc is declared in <stdlib.h>, so we include this header file in any program that calls malloc. The malloc function reserves a block

of memory of specified size and returns a pointer of type `void`. This means that we can assign it to any type of pointer. The general syntax of `malloc()` is

```
ptr =(cast-type*)malloc(byte-size);
```

where `ptr` is a pointer of type `cast-type`. `malloc()` returns a pointer (of cast type) to an area of memory with size `byte-size`.

For example,

```
arr=(int*)malloc(10*sizeof(int));
```

This statement is used to dynamically allocate memory equivalent to 10 times the area of `int` bytes. On successful execution of the statement the space is reserved and the address of the first byte of memory allocated is assigned to the pointer `arr` of type `int`.

Programming Tip

To use dynamic memory allocation functions, you must include the header file stdlib.h.

`calloc()` function is another function that reserves memory at the run time. It is normally used to request multiple blocks of storage each of the same size and then sets all bytes to `zero`. `calloc()` stands for contiguous memory allocation and is primarily used to allocate memory for arrays. The syntax of `calloc()` can be given as:

```
ptr=(cast-type*) calloc(n,elem-size);
```

The above statement allocates contiguous space for `n` blocks each of size `elem-size` bytes. The only difference between `malloc()` and `calloc()` is that when we use `calloc()`, all bytes are initialized to zero. `calloc()` returns a pointer to the first byte of the allocated region.

When we allocate memory using `malloc()` or `calloc()`, a `NULL` pointer will be returned if there is not enough space in the system to allocate. A `NULL` pointer, points definitely nowhere. It is a *not a pointer* marker; therefore, it is not a pointer you can use. Thus, whenever you allocate memory using `malloc()` or `calloc()`, you must check the returned pointer before using it. If the program receives a `NULL` pointer, it should at the very least print an error message and exit, or perhaps figure out some way of proceeding without the memory it asked for. But in any case, the program cannot go on to use the `NULL` pointer it got back from `malloc()`/`calloc()`.

A call to `malloc`, with an error check, typically looks something like this:

```
int *ip = malloc(100 * sizeof(int));
if(ip == NULL)
{
    printf("\n Memory could not be allocated");
    return;
}
```

Write a program to read and display values of an integer array. Allocate space dynamically for the array.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, n;
    int *arr;
    printf("\n Enter the number of elements ");
    scanf("%d", &n);
    arr = (int*)malloc(n * sizeof(int));
    if(arr == NULL)
    {
        printf("   \n Memory Allocation Failed");
        exit(0);
    }
```

```
    for(i=0;i<n;i++)
    {
        printf("\n Enter the value %d of the array: ", i);
        scanf("%d",&arr[i]);
    }
    printf("\n The array contains \n");
    for(i=0;i<n;i++)
        printf("%d", arr[i]);
    return 0;
}
```

Now let us also see how we can allocate memory using the `calloc` function. The `calloc()` function accepts two parameters—`num` and `size`, where `num` is the number of elements to be allocated and `size` is the size of elements. The following program demonstrates the use of `calloc()` to dynamically allocate space for an integer array.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int i,n;
    int *arr;
    printf ("\n Enter the number of elements: ");
    scanf("%d",&n);
    arr = (int*) calloc(n,sizeof(int));
    if (arr==NULL)
        exit (1);
    printf("\n Enter the %d values to be stored in the array", n);
    for (i = 0; i < n; i++)
     scanf ("%d",&arr[i]);
    printf ("\n You have entered: ");
    for(i = 0; i < n; i++)
        printf ("%d",arr[i]);
    free(arr);
    return 0;
}
```

Releasing the Used Space

When a variable is allocated space during the compile time, then the memory used by that variable is automatically released by the system in accordance with its storage class. But when we dynamically allocate memory then it is our responsibility to release the space when it is not required. This is even more important when the storage space is limited. Therefore, if we no longer need the data stored in a particular block of memory and we do not intend to use that block for storing any other information, then as a good programming practice we must release that block of memory for future use, using the `free` function. The general syntax of the `free()function` is,

```
free(ptr);
```

where `ptr` is a pointer that has been created by using `malloc()` or `calloc()`. When memory is de-allocated using `free()`, it is returned back to the free list within the heap.

To Alter the Size of Allocated Memory

At times the memory allocated by using `calloc()` or `malloc()` might be insufficient or in excess. In both the situations we can always use `realloc()` to change the memory size already allocated by `calloc()` and `malloc()`. This process is called *reallocation of memory*. The general syntax for `realloc()` can be given as,

```
ptr = realloc(ptr,newsize);
```

The function `realloc()` allocates new memory space of size specified by `newsize` to the pointer variable `ptr`. It returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region. Thus, we see that `realloc()` takes two arguments. The first is the pointer referencing the memory and the second is the total number of bytes you want to reallocate. If you pass zero as the second argument, it will be equivalent to calling `free()`. Like `malloc()` and `calloc()`, `realloc` returns a void pointer if successful, else a `NULL` pointer is returned.

If `realloc()` was able to make the old block of memory bigger, it returns the same pointer. Otherwise, if `realloc()` has to go elsewhere to get enough contiguous memory then it returns a pointer to the new memory, after copying your old data there. However, if `realloc()` cannot find enough memory to satisfy the new request at all, it returns a null pointer. So again you must check before using that the pointer returned by the `realloc()` is not a null pointer.

```
/*Example program for reallocation*/
#include < stdio.h>
#include < stdlib.h>
#define NULL 0
int main()
{
    char *str;
    str = (char *)malloc(10);
    if(str==NULL)
    {
        printf("\n Memory could not be allocated");
        exit(1);
}
    strcpy(str,"Hi");
    printf("\n STR = %s", str);
    /*Reallocation*/
    str = (char *)realloc(str,20);
    if(str==NULL)
    {
        printf("\n Memory could not be reallocated");
        exit(1);
}
printf("\n STR size modified\n");
printf("\n STR = %s\n", str);
strcpy(str,"Hi there");
printf("\n STR = %s", str);
/*freeing memory*/
free(str);
return 0;
}
```

Note With realloc(), you can allocate more bytes without losing your data.

Dynamically Allocating a 2-D Array

We have seen how `malloc()` can be used to allocate a block of memory which can simulate an array. Now we can extend our understanding further to do the same to simulate multidimensional arrays.

If we are not sure of the number of columns that the array will have, then we will first allocate memory for each row by calling `malloc`. Each row will then be represented by a pointer. Look at the code below which illustrates this concept.

```
#include <stdlib.h>
#include <stdio.h>
```

```
int main()
{
    int **arr,i,j,ROWS,COLS;
    printf("\n Enter the number of rows and columns in the array: ");
    scanf("%d %d",ROWS,COLS);
    arr = (int **)malloc(ROWS * sizeof(int *));
    if(arr == NULL)
    {
        printf("\n Memory could not be allocated");
        exit(-1);
    }
    for(i=0; i<ROWS; i++)
    {
        arr[i] = (int *)malloc(COLS * sizeof(int));
        if(arr[i] == NULL)
        {
            printf("\n Memory Allocation Failed");
            exit(-1);
        }
    }
    printf("\n Enter the values of the array: ");
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
            scanf("%d",&arr[i][j]);
    }
    printf("\n The array is as follows: ");
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
            printf("%d",arr[i][j]);
    }
    for(i = 0; i < ROWS; i++)
        free(arr[i]);
    free(arr);
    return 0;
}
```

Here, `arr` is a pointer-to-pointer-to-`int`: at the first level as it points to a block of pointers, one for each row. We first allocate space for rows in the array. The space allocated to each row is big enough to hold a `pointer-to-int`, or `int *`. If we successfully allocate it, then this space will be filled with pointers to columns (number of `int`s). This can be better understood from Fig. A2.

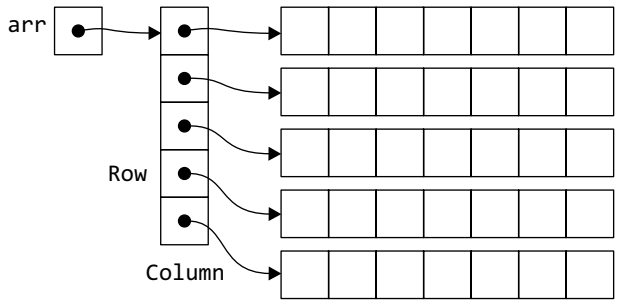

Figure A2 Memory allocation of two-dimensional array

Once the memory is allocated for the two-dimensional array, we can use subscripts to access its elements. When we write, `arr[i][j]`, it means we are looking for the `i`th pointer pointed to by `arr`, and then for the `j`th `int` pointed to by that inner pointer.

When we have to pass such an array to a function, then the prototype of the function will be written as

```
func(int **arr, int ROWS, int COLS);
```

In the above declaration, `func` accepts a `pointer-to-pointer-to-int` and the dimensions of the arrays as parameters, so that it will know how many rows and columns are there.